Multiparty Session C: Safe Parallel Programming with Message Optimisation

Nicholas Ng*, Nobuko Yoshida*, and Kohei Honda†

*Imperial College London †Queen Mary, University of London

Abstract. This paper presents a new efficient programming toolchain for message-passing parallel algorithms which can fully ensure, for any typable programs and for any execution path, deadlock-freedom, communication safety and global progress through a static checking. The methodology is embodied as a multiparty session-based programming environment for C and its runtime libraries, which we call Session C. Programming starts from specifying a *global protocol* for a target parallel algorithm, using a protocol description language. From this global protocol, the projection algorithm generates *endpoint protocols*, based on which each endpoint C program is designed and implemented with a small number of concise session primitives. The endpoint protocol can further be refined to a more optimised protocol through subtyping for asynchronous communication, preserving original safety guarantees. The underlying theory can ensure that the complexity of the toolchain stays in polynomial time against the size of programs. We apply this framework to representative parallel algorithms with complex communication topologies. The benchmark results show that Session C performs competitively against MPI.

1 Introduction

High-performance computing based on message-passing is one of the highly scalable frameworks for executing parallel algorithms with a wide range of hardware configurations starting from a small LAN to a large cluster to supercomputers. It is, however, hard to implement message-passing applications correctly, partly because they rely on not only local calculation at each endpoint, but also on global message exchange among all endpoints: if the message-passing part of a program is wrongly implemented, then the result of the calculation is either unavailable (e.g. by deadlock) or wrong (e.g. by receiving some values at wrong timings or as wrong types), even if all local calculations are correct.

One of the root causes of errors in communications programming is the lack of conformance to an assumed protocol among endpoint programs. Typical examples (written as MPI commands [27]) are a circular wait such as `MPI_Send(to2)` from process `1`, `MPI_Recv(from3)` from process `2` and `MPI_Send(to1)` from process `3`; and a communication mismatch such as `MPI_Recv(from2)` followed by `MPI_Send(to3)` from process `1`, `MPI_Recv(from3)` followed by `MPI_Send(to1)` from process `2` and `MPI_Recv(from1)` followed by `MPI_Send(to2)` from process `3`. To avoid such deadlocks,

one might permute the order of messages using asynchronous sending such as `Isend` followed by `Recv`, but it is often forgotten to write a required synchronisation (`Wait`). These are simple errors often illustrated in the textbooks [14, 15], but still appeared in many programs including large scale MPI applications, e.g [24]. Such communication errors are often hard to detect except by runtime analysis. Even if detected, hard to locate and fix the bug because the issue comes from distributed processes. Testing in general does not offer full safety assurance as it relies on executing a particular sequence of events and actions.

This paper proposes a new programming framework for message-passing parallel algorithms centring on explicit, formal description of *global protocols*, and examines its effectiveness through an implementation of a toolchain for C. All validations in the toolchain are done statically and are efficient, with a polynomial-time bound with respect to the size of the program and global protocol. The framework is based on theory of multiparty session types [3, 10, 18], and it supports a full guarantee of deadlock-freedom, type-safety, communication-safety and global progress for any well-typed programs. Global protocols serve as a guidance for a programmer to write safe programs, representing a type abstraction of expressive communication structures (such as sequencing, choice, broadcast, synchronisation and recursion). The toolchain uses a language Scribble [16, 31] for describing the multiparty session types in a Java-like syntax.

```
protocol Simple(role P1, role P2, role P3) {
  int   from P1 to P2;
  char  from P3 to P1;
  float from P2 to P3;
}
```

A simple example of a protocol in Scribble which corrects the first erroneous MPI program (a wait cycle) is given on the left. For endpoint code development, the programmer uses the *endpoint protocol* generated by the projection algorithm in the toolchain. For example, the above global protocol is projected to `P2` to obtain `int from P1; float to P3;`, which gives a template for developing a safe code for `P2` as well as a basis of static verification. Since we start from a correct protocol, if endpoint programs conform to the induced endpoint protocols, it automatically ensures deadlock-free, well-matched interactions.

Overview of the toolchain. A Session C program is developed in a top-down approach. Fig. 1 (l.h.s.) shows the relationships between the four layers (i–iv) that make up a complete Session C program. A Session C programmer first designs a global protocol (i) using Scribble (explained in § 2.1). A Session C program is a collection of individual programs (iv) in which each of the programs implements a participant (called *endpoint*) of the communication. We first extract the endpoint protocol from the global protocol by *projection* (ii). The projection takes the global protocol G and an endpoint (say Alice), and extracts only the interaction that involves Alice (T_{Alice}). Step (iii) describes a key element of our toolchain, the *protocol refinement*. T'_{Alice} is an endpoint protocol refined from the original T_{Alice}. This allows the programmer to write a more refined program P_{Alice} (which conforms to T'_{Alice}) than a program following the original T_{Alice}. Session C supports the *asynchronous message optimisation* [25, 26], the reordering of messages for minimising a waiting time as a refinement, through its subtyping checker (§ 3.2). Once P_{Alice} conforms T'_{Alice} such that $T'_{\text{Alice}} < T_{\text{Alice}}$

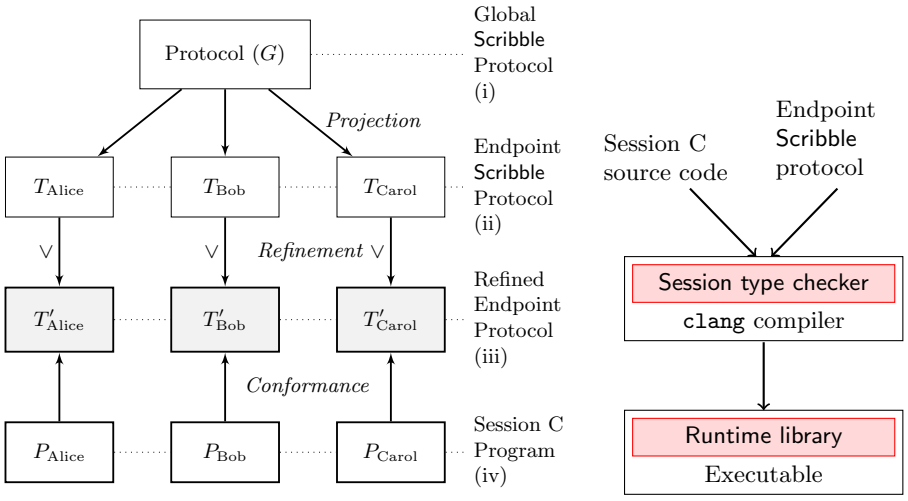


Fig. 1. Session C programming framework (l.h.s.) and architecture (r.h.s.).

(T'_{Alice} is more refined), then P_{Alice} automatically enjoys safety and progress in its interactions with P_{Bob} and P_{Carol}.

Programming environment. The programming environment of Session C is made up of two main components, the *session type checker* and the *runtime library* (§ 2.2). Fig. 1 (r.h.s.) shows the architecture. The session type checker takes an endpoint protocol (T_{Alice}) and a source code P_{Alice} as an input from the user. The endpoint protocol is generated from the global protocol G through the projection algorithm. The session type checker validates the source code against its endpoint protocol. When the program is optimised, it generates T'_{Alice} from P_{Alice} and checks if $T'_{\text{Alice}} < T_{\text{Alice}}$ (§ 3.2). The API provides a simple but expressive interface for session-based communications programming.

Contributions.

1. A toolchain for developing and executing message-passing parallel algorithms based on a formal and explicit description of interaction protocols (§ 2.1), with an automatic safety guarantee. All algorithms used in the toolchain are polynomial-time bounded (§ 3.2).
2. The first multiparty session-based programming environment for a low-level language, Session C, built from expressive session constructs supporting collective operations (§ 2), together with the associated runtime library.
3. A session type checker for Session C, which is the first to offer automatic, full formal assurance of communication deadlock-freedom (i.e. for any possible control path and interleaving) for a large class of message-passing parallel programs (§ 3.1), supporting messaging optimisations through the incorporation of the asynchronous subtyping [25, 26] (§ 3.2).
4. The practical validation of our methodology through the implementations of typical message-passing parallel algorithms, leading to concise and clear

programs (§ 4). The benchmark results show that representative parallel algorithms in Session C are executed competitively against their counterparts in MPI (the overhead is on average 1%) (§ 5).

All code and details of benchmark results are available from [13].

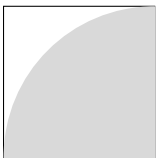
Acknowledgements. We thank Gary Brown for his fantastic work under the Scribble project and the members of Mobility Reading Group and Custom Computing Group for their discussions. The work is partially supported by EPSRC EP/F003757/01 and EP/G015635/01.

2 Protocols and Programming in Session C

2.1 Scribble, a protocol description language

Our toolchain uses Scribble [16, 31], a developer-friendly notation for specifying application-level communication protocols based on the theory of multiparty session types [3, 10, 18]. Scribble's development tool [31] supports parsing, well-formedness checking and endpoint projection, with bindings to multiple programming languages. We briefly introduce its syntax.

```
1 /* Protocol: Monte Carlo Pi estimation. */
2 import int;
3 protocol MonteCarloPi(role Master, role Worker0, role Worker1) {
4   // number of simulations to do in each worker
5   int from Master to Others; // broadcast
6   rec LOOP {
7     from Others to Master { Yes: No: }; // gather
8     LOOP; }
9 }
```



Above listing shows a simple Scribble global protocol for Monte Carlo π estimation. The algorithm uses random sampling to estimate the value of π. A Scribble protocol begins with the preamble, in Line 1, consisting of a message type declaration after the keyword `import`. Then the protocol definition is given starting from, in Line 2, the keyword `protocol`, followed by the protocol name, `MonteCarloPi`, and its parameters which are the roles to be played by participants. Then the description of a conversation structure follows. Line 4 says that the `Master` should send an integer (which specifies the number of tries) to `Others`, i.e. to all other roles than `Master`, i.e. to the workers. Line 5 declares a recursion named `loop`. In Line 6, (after each worker locally generates a random point on a square and tests if the point is in the quarter of a circle, i.e. the shaded area in the right figure above. `Master` is informed by `Others` (workers) whether the test was a hit, by choosing `Yes` or `No`. Regardless, Line 7 recurs.

The description of interaction in Lines 4-8 is generic, catering for any number of workers. Here we use *collective roles* in Scribble, where a single role can denote multiple participants. We introduce two collectives roles, `All` (for "every role") and `Others` (for "all other roles"). Using them, we can accurately represent the protocols for MPI collective operations as:

– `MPI_Bcast` (broadcast) from `A`: `from A to Others`.

– `MPI_Reduce` to `A`, a gather operation: `from Others to A`.
– `MPI_Barrier` with `A` as a gather point, for which we use consecutive interactions: `from Others to A`; `from A to Others`.
– `MPI_Alltoall`, a scatter-gather operation: `from All to Others`.

These collective roles can be used as a source and/or a target as far as it is not ambiguous (e.g. `from Others to Others`) and it does not induce a self-circular communication (e.g. `from All to All`). Each `All` is macro-expanded for each endpoint when projecting a global protocol, whereas `Others` is preserved after projection and is linked to programming constructs, as we shall discuss later.

Global protocol	Endpoint protocol
U `from myrole to role1,.., rolen/Others`	U `to role1,.., rolen/Others`
U `from role1,.., myrole,.., rolen/Others to role`	U `to role`
U `from role1,.., rolen/Others to myrole`	U `from role1,..,rolen/Others`
U `from role to role1,.., myrole,.., rolen/Others`	U `from role`
U `from All to Others`	U `to Others`; U `from Others`
`from myrole to role {` $l_1 : T_1 \cdots l_n : T_n$ `}`	`to role {` $l_1 : T'_1 \cdots l_n : T'_n$ `}`
`from role to myrole {` $l_1 : T_1 \cdots l_n : T_n$ `}`	`from role {` $l_1 : T'_1 \cdots l_n : T'_n$ `}`
`from All to Others {` $l_1 : T_1 \cdots l_n : T_n$ `}`	`to Others {` $l_1 : T'_1 \cdots l_n : T'_n$ `};` `from Others {` $l_1 : T'_1 \cdots l_n : T'_n$ `}`
`repeat from myrole to role {` T `}`	`repeat to role {` T' `}`
`repeat from role to myrole {` T `}`	`repeat from role {` T' `}`
`rec X {` T `}`	`rec X {` T' `}`

We present a summary of the Scribble syntax for global and local protocols in above table, which also shows how the former is projected to the latter. In each line, the left-hand side gives a syntax of a global protocol, while the right-hand side gives its projection onto participant `myrole`. U is a payload type; T and T' are global and endpoint types; and l is a label for branching. T and T' can be empty, denoting termination. Line 1 indicates two cases, one "U `from myrole to role1,.., rolen`", which is a multicast from `myrole` to n other roles; and "U `from myrole to Others`", which is a multicast from `myrole` to all others. Similarly for Lines 2-4. The right-hand side views the left-hand global interaction from the viewpoint of `myrole`. In Line 5, "`from All to Others`" means "every role sends to the remaining roles". Hence, for `myrole`, it means (1) it is sending to all others, i.e. broadcast; and then (2) receiving from all others, i.e. reduce.

```
/* Endpoint Scribble for Master */
import int;
protocol MonteCarloPi at Master
    (role Worker0, role Worker1){
  int to Others;
  rec LOOP {
    from Worker0, Worker1 { Yes: No: }
  LOOP; }
}
```

```
/* Endpoint Scribble for Worker0 */
import int;
protocol MonteCarloPi at Worker0
    (role Master, role Worker1) {
  int from Master;
  rec LOOP {
    to Master { Yes: No: }
  LOOP; }
}
```

As a concrete example of projection acting on the whole protocol, the endpoint protocols resulting from the projection of the Monte Carlo simulation example onto `Master` and `Worker0`, respectively, are given in the above listings.

2.2 Session C: programming and runtime

Session C offers a high-level interface for safe communications programming based on a small collection of primitives from the session type theory. These primitives are supported by a runtime whose implementation currently uses the ØMQ (ZeroMQ) [37] socket library, which provides efficient messaging over multiple transports including local in/inter-process communication, TCP and PGM (Pragmatic General Multicast).

A Session C program is a C program that calls the session runtime library. The following code implements `Master` whose endpoint protocol is given in the previous subsection. In the `main` function, `join_session` (Line 7) indicates the start of a session, whose arguments (from the command line arguments `argc` and `argv`) are a session handle of type `session *` and the location of the endpoint Scribble file. `join_session` establishes connections to other participating processes in the session, according to a connection configuration information such as the host/port for each participant, automatically generated from the global protocol. Next, the lookup function `get_role` returns the participant identifier of type `role *`. Then we have a series of session operations such as `send_`*type* or `recv_`*type* (discussed below). Lines 15-18 expand the choice from `Others` in the protocol into individual choices. Finally an `end_session` cleans up the session. Any session operation before `join_session` or after `end_session` is invalid because they do not belong to any session.

```
1  /* Session C implementation for Master */
2  #include <libsess.h>
3  ...
4  int main(int argc, char *argv[])
5  {  // variable declaration ...
6    session *s;
7    join_session(&argc, &argv, &s, "MCPi_Master.spr");
8    const role *Worker0 = s->get_role(s, "Worker0");
9    const role *Worker1 = s->get_role(s, "Worker1");
10
11   int count = 100;
12   msend_int(100, _Others(s));
13
14   while (count-- > 0) {
15     switch(inbranch(Worker0, &rcvd))
16         { case Yes: hits++; break; case No: break; }
17     switch(inbranch(Worker1, &rcvd))
18         { case Yes: hits++; break; case No: break; }
19   }
20   printf("Pi: %.5f\n", (4*hits)/(2*100.0));
21   end_session(s);
22 }
```

Programming Communications in Session C. We now outline communication primitives of Session C. In addition to the standard send/receive primitives, our library includes a primitive for multicast sending and its reverse. `msend` sends the same value to all receivers, and `mrecv` receives values (not necessarily identical but of the same type) from multiple senders, as we illustrate below.

The table above lists these primitives as well as control primitives we illustrate next, in correspondence with the Scribble protocol construct introduced in the § 2.1. The first six lines are for message passing. Each function name mentions a type explicitly, as in `send_`*datatype*, following MPI and to ensure type-safety under the lack of strong typing in C. We support `char`, `int`, `float`, `double`, `string` (C-string, contiguous NULL-terminated array of `char`), `int_array` (contiguous ar-

Scribble endpoint	Session C runtime interface
`int to Bob`	`send_int(role *r, int val);`
`string from Bob`	`recv_string(role *r, char *str);`
`int to role1,..,rolen`	`msend_int(int val, int roles_count,...);`
`string from role1,..,rolen`	`mrecv_string(char *str, int roles_count,...);`
`int to Others`	`msend_int(int val, _Others(sess));`
`string from Others/role1,..,rolen`	`mrecv_string(char *str, _Others(sess));`
`repeat to Bob { ... }`	`while(outwhile(int cond,int roles_cnt,...)){..}`
`repeat from Bob { ... }`	`while(inwhile(int roles_cnt, ...)){..}`
`rec { ... }`	ordinary `while` loop or `for` loop
`to Bob { LABEL0: ... }`	`outbranch(role *r, const int label);`
`from Bob { LABEL0: ... }`	`inbranch(role *r, int *label);`

ray of `int`), `float_array` (contiguous array of `float`), and `double_array` (contiguous array of `double`). These types are sufficient for implementing most parallel algorithms; for composite types that are not in the runtime library, the programmer can choose to combine existing primitives, or augment the library with marshalling and unmarshalling of the composite type, to allow type checking.

In Lines 3/4 of the table, `msend` and `mrecv` specify the number of roles (a roles count) of the targets/sources, respectively. Lines 5/6 show how the programmer can specify `Others` in `msend` and `mrecv`: the roles count and roles list are replaced by a macro `_Others(s)` with the session handle as the argument.

Structuring message flows: branching and iteration. Branching (choice) in Session C is declared explicitly by the use of `outbranch` and `inbranch`. Different branches may have different communication behaviours, and the deciding participant needs to inform the other participant which branch is chosen. The passive participant will then react accordingly.

```
if (i>3) {
  outbranch(Bob, BR_LABEL0);
  send_int(Bob, 42);
} else {
  outbranch(Bob, BR_LABEL1);
  recv_int(Bob, &val);
}
```

```
switch (inbranch(Alice, &rcvd_label)) {
  case BR_LABEL0:
    recv_int(Alice, &val);
    break;
  case BR_LABEL1:
    send_int(Alice, 42);
    break;}
```

Above, the branching is initiated by a call to `outbranch` in the then-block or else-block of an if-statement. On the receiving side of the branch, the program first calls `inbranch` to receive the branch label. A switch-case statement should then be used to run the segment of code which corresponds to the branching label.

For iteration, two methods are provided: *local* and *communicating iterations*. *Local iteration* is a standard statement such as `while`-statements, with session operations occurring inside. *Communicating iteration* is a distributed version of loop, where, at each iteration, the loop condition is computed by the process calling `outwhile` and is communicated to processes calling `inwhile`. This while loop is designed to support multicast, so that a single `outwhile` can control multiple processes. This is useful in a number of parallel iterative parallel algorithms, which the loop continues until certain conditions (e.g. convergence) are reached and cannot be determined statically.

```
// Master process (Alice)
while (outwhile(i++<3, 1, Bob))
  recv_int(Bob, &value);
```

```
// Slave process (Bob)
while (inwhile(1, Alice))
  send_int(Alice, 42);
```

Above, `Alice` issues an `outwhile` with condition `i++<3` which will be evaluated in each iteration. `outwhile` then sends the result of the evaluation (i.e. `1` or `0`) to `Bob` and also uses that as the local `while` loop condition. Then `Bob` receives the result of the condition evaluation from `Alice` by the `inwhile` call, and uses as the local `while` loop condition. Both processes execute the body of the loop, where `Bob` sends an integer to `Alice`. This repeats until `i++<3` evaluates to `0`, then both processes exit the while loop.

3 Type checking and message optimisation

3.1 Session type checker

The session type checker for an endpoint program is implemented as a `clang` C compiler plugin. The `clang` compiler is the full-featured C/C++/Objective-C compiler frontend of the LLVM (Low-Level Virtual Machine) project [22]. LLVM is a collection of modular and reusable individual libraries for building compiler toolchains. The modular approach of the project allows easy mix-and-match of individual components of a compiler to build source code analysis and transformation tools. Our session type checker is built as such a tool, utilising the parser and various AST-related frontend modules from the `clang` compiler.

Endpoint type checking verifies that the source code conforms to the corresponding endpoint protocol in Scribble. The type checker operates by ensuring that the linear usage of the communication primitives conforms to a given Scribble protocol, based on the correspondence between Scribble and Session C constructs given in the table in § 2.2. The following example shows how Scribble statements are matched against Session C communication primitives.

We quickly outline how the type checker works, which also gives the background for §3.2 later. First, the Scribble endpoint protocol is parsed into an internal tree representation. For brevity, hereafter we refer to it as *session tree*. Except for recursion (which itself is *not* a communication), each node of a session tree consists of (1) the target role, (2) the type of the node (e.g. send, receive, choice, etc.) and (3) the datatype, if relevant (e.g. `int`, `string`, etc.). For example, a Scribble endpoint type statement "`int to Worker;`" becomes a node {`role: Worker, type: send, datatype: int`}.

The type checking is done by *inferring* the session typing of each program and matching the resulting session tree against the one from the endpoint protocol. The type inference is efficiently done by extracting session communication operations from the source code.[1] A session tree is then constructed from this

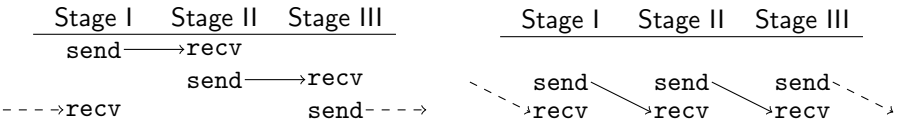
[1] Because C allows unrestricted type conversion by casting, we use the datatype explicitly mentioned in communication functions as the type of an argument rather than the type of its expression. For example, `send_int(Bob, 3.14)` says that sending 3.14 as `int` is the intention of the programmer, which is safe if the receiver is intended to receive an integer.

session typing. For example, a runtime function call, `send_int(Worker, result)` will be represented by a node {`role: Worker, type: send, datatype: int`}.

We can now move to the final process of session type checking in Session C. After their construction, the session trees from both Scribble endpoint protocol and the program are *normalised*, removing unused dummy nodes, branches without session operations and iteration nodes without children, thus compacting the trees to a canonical form. We then compare these two normalised session trees, and verify that they are in the asynchronous subtyping relation (illustrated in § 3.2) up to minimisation.

3.2 Asynchronous message optimisation

This subsection illustrates one of the key contributions of our toolchain, the type checking in the presence of *asynchronous message optimisation*. Parallel programs often make use of parallel pipelines to overlap computation and communication. The overlapping can reduce stall time due to blocking wait in the asynchronous communication model, as far as the overlapping does not interfere with data dependencies.



Above (left) shows a native but immediately safe ring pipeline and (right) an efficient parallel pipeline, which needs only two steps to complete instead of three, since Stage I does not need to wait for data from Stage III. However, this parallel pipeline is hard to type check against a naturally specified global type (which would be based on the left figure where interactions take place one by one), because of the permuted communication operations – we cannot match the `send` against the `recv`, because they criss-cross. But these two figures are equivalent under the asynchronous communication model with non-blocking send and blocking receive.

```
while (i++ < N) { /* StageII */
  recv_int(StageI, &rcvd);
  send_int(StageIII, result);
  compute(result);
  result = rcvd;
}
```

```
while (i++ < N) { /* Optimised StageII */
  send_int(StageIII, result);
  compute(result);
  recv_int(StageI, &rcvd);
  result = rcvd;
}
```

To see this point concretely, the above listing juxtaposes an unoptimised and optimised implementation of the Stage II. Both programs communicate values correctly despite the different order of communication statements. Note `compute` is positioned after a send, so that `compute` can be carried out while the data is being sent in the background, taking advantage of non-blocking sends.

The use of parallel pipelines is omnipresent in message-passing parallel algorithms. To type-check them, we apply the asynchronous subtyping theory [25, 26], which allows the following deadlock-free permutations:

1. Permuting Receive-Send to Send-Receive in the same or different channels;

2. Permuting order of Send-Send if they are in difference channels;
3. Permuting order of Receive-Receive if they are in different channels

Note that if we permute in the different direction from (1) (i.e. Send-Receive to Receive-Send), it causes a deadlock. E.g. in the efficient pipeline described above, if `send-recv` is permuted to `recv-send` in the Stage I, it causes a deadlock between the Stage I and II.

We give the subtyping rules against Scribble endpoint protocols below, taking the iso-recursive approach [25], where T is an endpoint type: where use the type

$$\frac{-}{T < T}\lfloor \textsc{Id} \rfloor \quad \frac{\forall i.\ T_i < T_i'}{\texttt{from/to role}\ \{l_1 : T_1 \cdots l_n : T_n\} < \texttt{from/to role}\ \{l_1 : T_1' \cdots l_n : T_n'\}}\lfloor \textsc{Bra} \rfloor$$

$$\frac{T_1 < C[T_2] \quad U'\ \texttt{to role} \notin C \quad \forall \texttt{role}'.\ U'\ \texttt{from role}' \notin C}{U\ \texttt{from role}; T_1 <\ C[U\ \texttt{from role}; T_2]}\lfloor \textsc{Recv} \rfloor$$

$$\frac{T_1 < C[T_2] \quad U'\ \texttt{from/to role} \notin C}{U\ \texttt{to role}; T_1 <\ C[U\ \texttt{to role}; T_2]}\lfloor \textsc{Send} \rfloor \quad \frac{T_1 < T_2}{\texttt{rec X}\,\{T_1\} < \texttt{rec X}\,\{T_2\}}\lfloor \textsc{Rec} \rfloor$$

context C defined as:

$$C\ ::=\ [] \ \mid\ U\ \texttt{from role}; C \ \mid\ U\ \texttt{to role}; C$$

The subtyping algorithm in Session C conforms to the rules listed above (which come from [25]) and is their practical refinement, which we describe below:

1. ($\lfloor \textsc{Recv} \rfloor$) For each receive statement, search for a matching receive for the same channel in the source code until a receive statement is found or search failed. Send and other statements in different channels can be skipped over.
2. ($\lfloor \textsc{Send} \rfloor$) For each send statement, search for a matching send for the same channel in the source code until a receive statement is found or search failed. Sends can only be permuted between statements in different channels, so overtaking a receive operation is disallowed.
3. We apply the permutation described above on consecutive statements within `rec` and `repeat` blocks following the iso-recursive approach [25], which is more suitable for languages such as C and Java.

Finally, we check that all nodes in the source code and protocol session type trees have been visited.

We end this section by identifying the time-complexity of the present toolchain. It uses well-formedness checking of a global protocol and its projection, which are both polynomial-time bound w.r.t. the size of the global type [8, 10]. The asynchronous subtype-checker as given above is polynomial against the size of a local type based on the arguments from [8, 25, 26]. Type inferences for session typed processes are polynomial [10, 18, 26]. We conclude:

Remark 1. The complexity of the whole toolchain is polynomial time-bounded against the size of a global type and a program.

Thus the toolchain is in principle efficient. Further, a careful examination of each algorithm suggests they tend to perform linearly with a small factor in normal cases (e.g. unless deeply nested permutations are carried out for optimisations). Our usage experience confirms this observation.

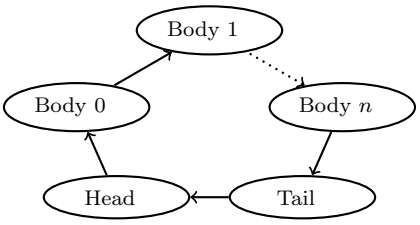
4 Parallel algorithms

In this section we demonstrate the effectiveness of Session C for clear, structured and safe message-passing parallel programming, through two algorithms which exemplify complex optimisations and communication topologies. For other implementations of representative parallel algorithms [11, 15, 23], see [13].

4.1 N-body simulation: asynchronous optimisation for pipelines

The parallel N-body algorithm forms a circular pipeline. Such a ring topology [2] is used in many parallel algorithms such as LU matrix decomposition [6]. The N-body problem involves finding the motion, according to classical mechanics, of a system of particles given their masses, initial positions and velocities. Parallelisation is achieved by partitioning the particle set among a set of m worker processes. Each worker is responsible for a partition of all particles.

```
protocol Nbody /* Global protocol */
    (role Head, role Body, role Tail) {
  rec NrOfSteps {
    rec SubCompute {
      particles from Head to Body;
      particles from Body to Tail;
      particles from Tail to Head;
      SubCompute; }
    NrOfSteps; }
}
```



Above shows the global protocol with 3 workers, `Head`, `Body` and `Tail`. The simulation is repeated for a number of steps (`rec NrOfSteps`). In each step, the resultant forces of particles held by a worker are computed against all particles held by others. We arrange our workers in a ring pipeline and perform a series of sub-computations (`rec SubCompute`) to propagate the particles to all workers, each involving receiving particles from a neighbouring worker and sending particles received in the previous sub-computation to the next worker.

```
protocol Nbody at Body /*endpoint*/
    (role Head, role Tail) {
  rec NrOfIters {
    rec SubCompute {
      particles from Head;
      particles to Tail;
      SubCompute;}

    NrOfIters;}
}
```

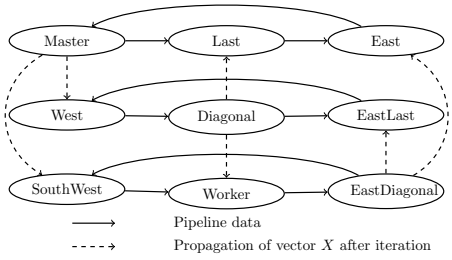
```
/* Implementation of Body worker */
while (iterations++ < NR_OF_ITERATIONS) {
  while (rounds++ < NR_OF_NODES) {
    send_particles(Tail, tmp_parts);//permuted
    // Update veclocities
    compute_forces(particles, tmp_parts,...);
    recv_particles(Head, &tmp_parts);
  } // Update positions by reeceived velocities
  compute_positions(particles, pvs, ... );
}
```

All of the endpoint protocols inherit the two nested `rec` blocks from the global protocol. In the body block of `rec SubCompute`, the order of send and receive are different in `Head` and `Body`. As discussed in §3.2, Session C allows permuting the order of send and receive for optimisations under the asynchronous subtyping, so that we can type-check this program. Using the endpoint protocols as specification, we can implement the workers. The code on the right implements the `Body` worker which is typable by our session type checker, despite the difference in order of send and receive from its endpoint Scribble.

4.2 Linear equation solver: a wraparound mesh topology

The aim of the linear equation algorithm is finding a x such that $Ax = b$, where A is an $n \times n$ matrix and x and b are vectors of length n. We use the parallel Jacobi algorithm [1], which decomposes A into a diagonal component D and a remainder R, $A = D + R$. The algorithm iterates until the normalised difference between successive iterations is less than a predefined error.

```
/* Global protocol */
protocol Solver (role Master, ...) {
  rec Iter {
    rec Pipe {
      double_array from Master to Last;
      double_array from Last to East;
      double_array from East ro Master;
      // Other communication in pipeline
      Pipe;}
    // Distribute X vector from diagonal
    double_array from Master to SouthWest;
    double_array from Master to West;
    // Distribution of other columns
    Iter;}
}
```



Our parallel implementation of this algorithm uses p^2 processors in a $p \times p$ wraparound mesh topology to solve an $n \times n$ system matrix. The matrix is partitioned into submatrix blocks of size n^2/p^2, assigned to each of the processors. Above shows the global protocol and the dataflow of the linear equation solver implementation with 9 workers.

An endpoint protocol is listed below on the left. The overall iteration of the algorithm is controlled by a `rec Iter` block. In each iteration, the computed values are put into a horizontal pipeline, as shown on the right to compute the sums. The resultant X vector is then calculated by the diagonal node to other workers in the mesh for the next iteration. The corresponding code is given on the right. The asynchronous message optimisation is again applied to the horizontal pipeline in order to overlap communications and computations.

```
protocol Solver at Diagonal
    (role West, role EastLast,
     role Last, role Worker) {
  rec Iter {
    rec Pipe {
      double_array from West;

      double_array to EastLast;
      Pipe;
    }
    double_array to Last, Worker;
    Iter;
  }
}
```

```
while (!iter_completed)) {
  computeProducts(partsum, blkA, newXVec, ...);
  computeSums(sum, partsum, ...);
  pipe = 0;
  while (pipe++ < columns) {
    send_double_array(EastLast, partsum, blkDim);
    computeSums(sum, partsum, blkDim);
    recv_double_array(West, partsum, &length);
  }
  // calculate X vector
  copyXVector(newXVec, oldXVec, ...);
  computeDivisions(newXVector, sum, ...);
  msend_double_array(newXVec, Last, Worker, ...);
}
```

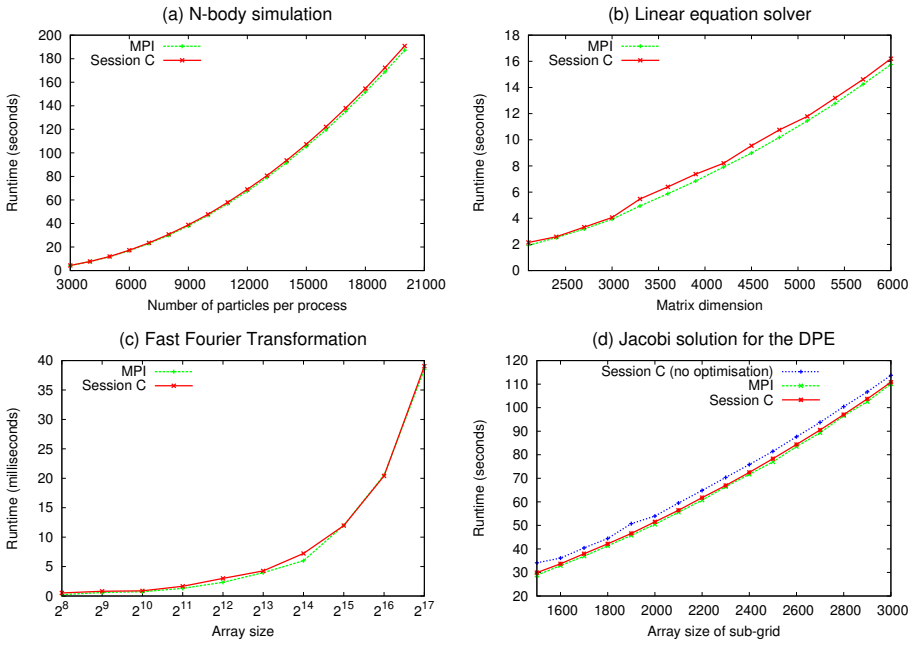


Fig. 2. Benchmark results.

5 Performance evaluation

This section presents performance results for several implementations of the four algorithms which feature different topologies and communication structures. The first three benchmarks were taken on workstations with Intel Core i7-2600 processors with 8GB RAM running Ubuntu Linux 11.04; the Jacobi solution benchmarks were taken on a high performance cluster with nodes containing AMD PhenomX49650 processor with 8GB RAM running CentOS 5.6, connected by a dedicated Gigabit Ethernet switch. Each benchmark was run 5 times and the reported runtime is the average of all 5 runs. For the MPI versions, OpenMPI 1.4.3 were used. Both use gcc 4.4.3 to compile with the optimisation level `-O3`.

N-body simulation. Our results are compared against MPI. Both versions use a ring pipeline to propagate the particles, and the two implementations share the same computational component by linking the same compiled object code for the `compute` functions. Our implementations were benchmarked with 3 workers and 1000 iterations which we perform a simulation on a set of input particles in the two-dimensional space. The results in Fig. 2(a) show that Session C's execution time is within 3% of the MPI implementation.

Linear equation solver. Fig. 2(b) shows that the MPI linear equation solver is faster than Session C implementation by 1–3%, with the ratio decreasing as the matrix size increases, suggesting the communication overhead is low, if any. The MPI implementation uses `MPI_Bcast` to broadcast the results of each iteration to

all nodes in the column, while Session C explicitly distributes the results.

FFT (Fast Fourier Transform) butterfly algorithm. We use a 8 node FFT butterfly. As seen from Fig. 2(c), Session C demonstrates a competitive performance compared to MPI implementation, again with the difference in ratio decreasing as the array size gets larger. The algorithm takes advantage of asynchronous optimisation for the butterfly message exchanges.

Jacobi solution for the discrete Poisson equation. Fig. 2(d) shows the benchmark results of the implementation of Jacobi solution. We benchmarked an optimised Jacobi solution implemented in Session C against a version without asynchronous message optimisation and found that the optimisation improved the performance by up to 8%. The result of this optimisation is very close (within 1%) to that of our reference implementation in MPI, demonstrating the effectiveness of the asynchronous optimisation.

6 Related works and further topics

Due to space limitations, we omit related works with session-based languages (such as Java, Haskell and OCaml) and HPC and PGAS languages, which are discussed in [29].

Deadlock detection in MPI. ISP [34] is a dynamic verifier which applies model-checking techniques to identify potential communication deadlocks in MPI (by "communication deadlock" we mean deadlocks due to communication mismatch/circularity, rather than local computation, e.g. divergent loop). Their tool uses a fixed test harness. In order to reduce the state space of possible thread interleavings of an execution, the tool exploits an independence between thread actions. Later in [35], they improved its scheduling policy to gain efficiency of the verification. TASS [32] is another model checking-based tool for a deadlock analysis in MPI. It constructs an abstract model of a given MPI program and uses symbolic execution to evaluate the model for finding deadlocks.

Our session type-based approach differs from these approaches in that it offers a full deadlock-free guarantee for communications by type-checked programs, without being restricted to external test sets or extracted models from program code, as well as offering a low-cost static checking. We believe a communication protocol is an abstraction which a developer of a message-passing parallel algorithm is anyway aware of. Session C encourages programmers to make this abstraction explicit, and offers primitives and a type checker for well-structured and formally safe message-passing parallel programs.

Formally-founded communication-based HPC languages. Pilot [5] is a parallel programming layer on top of standard MPI, aiming to simplify complex MPI primitives based on CSP. The communication is synchronous and channels are untyped to allow a reuse for different types. They have a runtime analysis for some deadlock patterns. Occam-pi [30] is a system-level efficient concurrent language with channel-based communication based on CSP and the π-calculus. It offers various locking and barrier abstractions, but do not support deadlock-

free analysis. Heap-Hop [33] is a verification tool for C based on dual contracts and Separation Logic. It can detect a deadlock based on contract specifications, but treats only *binary* (two parties) communications. Our work differs in that we centre on multiparty session-based abstractions for structured communications programming combined with a full formal assurance for communication safety.

Our previous work [29] applied Session Java (SJ) [19, 20], Java enhanced with session types, to parallel algorithms. SJ treats only *binary* session types [17] and cannot guarantee deadlock freedom and global progress between more than two processes. To ensure these properties, the tool in [29] has to run an additional topology verification on the top of the session type-checking. Session C offers a significant speed-up (60%) compared to SJ as well as MPI for Java [28].

Optimisation in MPI. Techniques for improving performance of MPI include building libraries for efficient transmission of data, e.g. [7] or MPI-aware optimising compilers, e.g. [12]. Most optimisations share a common theme to utilise computation and communication overlap to reduce the negative impact of the communication overhead. Our asynchronous message optimisation is one such instance to facilitate communication-computation overlap. Unlike Session C, existing works do not offer a similar framework, where a type-theoretic basis gives a formal safety assurance for optimised code.

Further topics include extending Scribble and the Session C programming framework to support parametrised [36] and multirole multiparty session types [9] to allow a fully generic protocol description and programming (e.g. with respect to the number of workers in the examples in §2); synthesis of global protocols for better development lifecycle; and add design-by-contracts [4] for more fine-grained logical verification. This work is limited to ensuring safety of communication. We intend to combine some features of Cyclone [21] to extend our framework to ensure some functional safety in addition to communication safety.

References

1. Jacobi and Gauss-Seidel Iteration. `http://math.fullerton.edu/mathews/n2003/GaussSeidelMod.html`
2. N-body algorithm using pipeline. `http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples/advmsg/nbodypipe_c.htm`
3. Bettini et al.: Global Progress in Dynamically Interleaved Multiparty Sessions. In: CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer (2008)
4. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: CONCUR. LNCS, vol. 6269, pp. 162–176 (2010)
5. Carter, J., Gardner, W.B., Grewal, G.: The Pilot approach to cluster programming in C. In: IPDPSW. pp. 1–8. IEEE (2010)
6. Casanova, H., Legrand, A., Robert, Y.: Parallel Algorithms. Chapman & Hall (Jul 2008)
7. Danalis, A., et al.: MPI-aware compiler optimizations for improving communication-computation overlap. In: ICS'09. pp. 316–325 (2009)
8. Deniélou, P.M., Yoshida, N.: Buffered Communication Analysis in Distributed Multiparty Sessions. In: CONCUR'10. LNCS, vol. 6269, pp. 343–357. Springer (2010)

9. Deniélou, P.M., Yoshida, N.: Dynamic multirole session types. In: POPL. pp. 435–446. ACM (2011)
10. Deniélou, P.M., Yoshida, N.: Multiparty session types meet communicating automata. In: ESOP. LNCS (2012), http://www.doc.ic.ac.uk/~malo/msa/
11. Dwarf Mine homepage. http://view.eecs.berkeley.edu/wiki/Dwarf_Mine
12. Friedley, A., Lumsdaine, A.: Communication Optimization Beyond MPI. In: IPDPSW and Phd Forum. IEEE (2011)
13. Online Appendix, http://www.doc.ic.ac.uk/~cn06/pub/2012/sessionc/
14. Grama, A., Karypis, G., Kumar, V., Gupta, A.: Introduction to Parallel Computing (2nd Edition). Addison Wesley, 2 edn. (Jan 2003)
15. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press (1999)
16. Honda, K., Mukhamedov, A., Brown, G., Chen, T.C., Yoshida, N.: Scribbling interactions with a formal foundation. In: ICDCIT. LNCS, vol. 6536, pp. 55–75. Springer (2011)
17. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Disciplines for Structured Communication-based Programming. In: ESOP. LNCS, vol. 1381, pp. 122–138. Springer-Verlag (1998)
18. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL'08. vol. 5201, p. 273 (2008)
19. Hu, R., Kouzapas, D., Pernet, O., Yoshida, N., Honda, K.: Type-Safe Eventful Sessions in Java. In: ECOOP. LNCS, vol. 6183, pp. 329–353 (2010)
20. Hu, R., Yoshida, N., Honda, K.: Session-Based Distributed Programming in Java. In: ECOOP. LNCS, vol. 5142, pp. 516–541 (2008)
21. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y.: Cyclone: A Safe Dialect of C. In: Usenix Annual Technical Conference, Monterey, CA (2002)
22. Lattner, C., Adve, V.S.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO'04. pp. 75–88 (2004)
23. Leighton, F.T.: Introduction to parallel algorithms and architectures: arrays, trees, hypercubes. Morgan Kaufmann (1991)
24. Metis and parmetis, glaros.dtc.umn.edu/gkhome/views/metis
25. Mostrous, D.: Session Types in Concurrent Calculi: Higher-Order Processes and Objects. Ph.D. thesis, Imperial College London (2009)
26. Mostrous, D., Yoshida, N., Honda, K.: Global principal typing in partially commutative asynchronous sessions. In: ESOP. LNCS, vol. 5502, pp. 316–332 (2009)
27. Message Passing Interface. http://www.mcs.anl.gov/research/projects/mpi/
28. MPJ Express homepage. http://mpj-express.org/
29. Ng, N., Yoshida, N., Pernet, O., Hu, R., Kryftis, Y.: Safe Parallel Programming with Session Java. In: COORDINATION. LNCS, vol. 6721, pp. 110–126 (2011)
30. Occam-pi homepage, http://www.occam-pi.org/
31. Scribble homepage, http://www.jboss.org/scribble
32. Siegel, S.F., Zirkel, T.K.: Automatic formal verification of MPI-based parallel programs. In: PPoPP'11. p. 309. ACM Press (Feb 2011)
33. Villard, J.: Heaps and Hops. Ph.D. thesis, ENS Cachan (2011)
34. Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R.M., Thakur, R.: Formal verification of practical MPI programs. In: PPoPP'09. pp. 261–270 (2009)
35. Vo, A., et al.: A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In: SC'10. pp. 1–10. IEEE (2010)
36. Yoshida, N., Deniélou, P.M., Bejleri, A., Hu, R.: Parameterised Multiparty Session Types. In: FOSSACS. LNCS, vol. 6014, pp. 128–145 (2010)
37. ZeroMQ homepage, http://www.zeromq.org/